📘 HBnB Evolution — Comprehensive Technical Documentation (Part 1)

---

0. High-Level Package Diagram

Overview

This package diagram illustrates the **three-layer architecture** of the HBnB Evolution system and how these layers communicate using the **Facade design pattern**. It provides a structural overview of responsibility separation, dependency direction, and interaction flow, ensuring maintainability, scalability, and testability.

The architecture enforces: - **Loose coupling** between layers - **Single responsibility per layer** - **Centralized orchestration** via the Facade

---

Architecture Layers

Layer 1: Presentation Layer

Responsibility: Handles all external communication with clients. This layer translates HTTP requests into structured service calls and formats domain responses into API-friendly outputs (JSON/HTTP responses).

Primary Responsibilities: - Input validation (schema, required fields, types) - Authentication and authorization checks - HTTP status code handling - Serialization/deserialization

Components: - **API Controllers (UserAPI, PlaceAPI, ReviewAPI, BookingAPI)** - Define RESTful endpoints - Map HTTP verbs to use cases (POST → create, GET → fetch, PUT → update, DELETE → remove)

- **Service Interfaces**
- Define contracts for business operations
- Ensure controllers depend on abstractions rather than implementations

Why this design? - Keeps controllers thin and logic-free - Allows API versioning and transport changes (e.g., REST → GraphQL) without impacting business logic

---

Layer 2: Business Logic Layer

Responsibility: Contains all **domain rules**, **validation logic**, and **workflow orchestration**. This layer models real-world business concepts and ensures system invariants are enforced regardless of entry point.

Primary Responsibilities: - Entity validation and lifecycle management - Enforcement of business rules - Coordination of multi-entity operations - Conflict resolution (e.g., booking overlaps)

Components: - **HbnbFacade** - Implements the Facade Pattern - Serves as the single entry point into the domain layer - Coordinates multiple entities and repositories

- **Domain Models (User, Place, Review, Amenity, Booking)**
- Encapsulate business state and rules
- Own validation logic and domain behaviors

Why this design? - Centralizes complex workflows - Protects domain integrity - Allows reuse of business logic across multiple interfaces (API, CLI, admin tools)

Layer 3: Persistence Layer

Responsibility: Manages **data storage, retrieval, and consistency** independently of business rules.

Primary Responsibilities: - CRUD operations - Query optimization - Transaction management - Data mapping between domain models and database schemas

Components: - **Repositories (UserRepository, PlaceRepository, BookingRepository, etc.)** - Abstract database access - Expose domain-friendly query methods

- **Database (PostgreSQL / MySQL)**
- Stores persistent state
- Enforces structural constraints

Why this design? - Enables database replacement without domain refactoring - Supports mocking/stubbing during unit testing

---

Facade Pattern Communication Flow

The **Facade Pattern** reduces coupling between the Presentation and Business Logic layers by exposing a **simple, unified API** to complex internal workflows.

Typical Request Flow

1. **Presentation Layer** calls a single method on `HbnbFacade`
2. **HbnbFacade** coordinates multiple domain entities and repositories
3. **Repositories** persist or retrieve data
4. **Facade** returns a structured domain result

Example

Instead of:

```
API → UserModel → PlaceModel → BookingModel → Repository
```

The system uses:

```
API → HbnbFacade → (internal orchestration) → Repository
```

Benefits

- Controllers remain simple
- Business workflows stay centralized
- Internal complexity is hidden
- Domain logic remains reusable

---

1. Detailed Class Diagram — Business Logic Layer

Overview

This section describes the **core domain entities** of HBnB Evolution and their relationships. These classes model real-world concepts (users, places, bookings, etc.) and encapsulate all business rules that ensure system correctness.

Each entity is responsible not only for storing state but also for enforcing invariants and validating operations on itself.

1.1 BaseEntity

Role

`BaseEntity` is an abstract superclass that provides shared identity and lifecycle behavior across all domain objects.

Key Attributes

- `id (UUID)` — Globally unique identifier for each entity
- `created_at (datetime)` — Timestamp of entity creation
- `updated_at (datetime)` — Timestamp of last modification

Key Methods

- `save()` — Persists the current state to storage
- `delete()` — Removes or soft-deletes the entity
- `update_timestamp()` — Automatically updates `updated_at`

Design Rationale

- Enforces consistent identity handling
- Eliminates duplication across domain classes
- Supports auditing and traceability

1.2 User Entity

Role

Represents all system users, including **guests**, **hosts**, and **administrators**. The User entity owns identity, authentication, authorization, and profile lifecycle responsibilities.

Key Attributes

- `email`
- `password_hash`
- `first_name`
- `last_name`
- `role (guest | host | admin)`
- `is_active`
- `email_verified`

Key Business Logic

- **Authentication:** `login()`, `logout()`, `change_password()`
- **Authorization:** `is_administrator()`, `can_host()`, `can_book()`
- **Profile Management:** `update_profile()`, `get_full_name()`
- **Account Lifecycle:** `register()`, `verify_email()`, `deactivate()`

Critical Business Rules

- Only **email-verified users** may create bookings
- **Inactive users** cannot authenticate
- **Administrative users** bypass certain restrictions
- A user must meet age and compliance requirements to host or book

---

1.3 Place Entity

Role

Represents properties available for rental. The Place entity manages publication state, pricing logic, availability windows, and amenity composition.

Key Attributes

- `name`
- `description`

- `price_per_night`
- `max_guests`
- `owner (User)`
- `status (draft | published | archived)`
- `location`

Key Business Logic

- **Lifecycle:** `create()`, `publish()`, `unpublish()`, `archive()`
- **Availability:** `check_availability(start, end)`
- **Pricing:** `calculate_price(start, end, guests, amenities)`
- **Amenities:** `add_amenity()`, `remove_amenity()`, `get_amenities()`
- **Reputation:** `get_reviews()`, `get_average_rating()`

Critical Business Rules

- Only **published places** appear in search results
- Price calculations factor:
- Seasonal rates
- Length-of-stay discounts
- Optional amenities
- Availability checks prevent overlapping bookings
- Only the **owner or admin** can modify listing state

1.4 Review Entity

Role

Represents feedback left by users after completing stays. Reviews affect place reputation and user trust scoring.

Key Attributes

- `rating`
- `comment`
- `user (User)`
- `place (Place)`
- `booking_id`
- `created_at`

Key Business Logic

- **Validation:** `validate()`, `is_within_review_period()`
- **Rating Aggregation:** `calculate_rating()` (cleanliness, accuracy, location, etc.)
- **Moderation:** `report()`, `can_edit()`, `flag_for_review()`

Critical Business Rules

- A review can only be submitted by a **user who completed a booking**
- Only **one review per booking** is allowed

- Reviews can only be edited within a defined time window
- Verified stays are weighted more heavily in rankings

1.5 Amenity Entity

Role

Represents features offered by places (WiFi, parking, pool, air conditioning, etc.). Amenities affect pricing, filtering, and guest decision-making.

Key Attributes

- `name`
- `category (kitchen | bathroom | outdoor | safety | etc.)`
- `is_standard`
- `is_active`

Key Business Logic

- **Categorization:** `get_category()`
- **Pricing Logic:** `is_included()`, `additional_cost()`
- **Association:** `get_places()`

Critical Business Rules

- Standard amenities are included in base price
- Premium amenities add surcharges
- Inactive amenities cannot be assigned to places
- Seasonal amenities may only be enabled during certain periods

---

1.6 Booking Entity

Role

Represents a reservation transaction between a guest and a place. Booking enforces availability, pricing, payment, and cancellation policies.

Key Attributes

- `start_date`
- `end_date`
- `guest_count`
- `status (pending | confirmed | cancelled | completed)`
- `user (User)`
- `place (Place)`
- `total_price`

Key Business Logic

- **Reservation:** `create()`, `confirm()`, `cancel()`
- **Validation:** `check_dates()`, `validate_guests()`, `is_cancellable()`
- **Pricing:** `calculate_total()`
- **Payments:** `process_payment()`, `refund()`

Critical Business Rules

- Check-in date must precede check-out date
- Guest count must not exceed place capacity
- Overlapping bookings are prohibited
- Cancellation rules depend on timing and policy type
- Refunds are calculated dynamically

---

2. Multiplicity Constraints — Detailed Explanation

This section defines **cardinality relationships** between domain entities and how they are enforced at both **database** and **business logic** levels.

2.1 User → Place (1 to Many)

Meaning

One User (host) may own and manage multiple Places. Each Place must have exactly one owner.

Database Enforcement

- `places.owner_id → users.id` (foreign key)
- Index on `owner_id` for fast retrieval
- Soft-delete or archival strategy instead of cascade delete

Business Logic Enforcement

- Place creation requires:
- User existence
- Active account
- Hosting eligibility
- Ownership transfer requires validation of new owner
- If a User is deactivated:
- Their Places are automatically unpublished
- Existing bookings may remain valid but future ones are blocked

---

2.2 Place → Review (1 to Many)

Meaning

A Place may accumulate multiple Reviews over time, each tied to a completed booking.

Database Enforcement

- `reviews.place_id → places.id` (foreign key)
- Index on `place_id`
- Trigger to update:
- `places.review_count`
- `places.average_rating`

Business Logic Enforcement

- Review creation requires:
- A completed booking for the same place
- Booking ownership by the reviewer
- Reviews are only accepted within a defined review window (e.g., 30 days)
- Duplicate reviews per booking are rejected
- Moderation pipelines flag abusive or suspicious content

2.3 User → Review (1 to Many)

Meaning

A User may submit many Reviews but only one per completed booking.

Database Enforcement

- `reviews.user_id → users.id` (foreign key)
- Composite unique constraint: `(user_id, booking_id)`
- Index on `user_id`

Business Logic Enforcement

- Identity verification (email, device trust, payment confirmation)
- Rate limiting (e.g., max 3 reviews/week)
- Reputation scoring adjusts review influence
- Users flagged for abuse may be restricted from reviewing

2.4 Place ↔ Amenity (Many to Many)

Meaning

A Place may offer many Amenities, and each Amenity may belong to many Places.

Database Enforcement

- Junction table: `place_amenities(place_id, amenity_id, additional_cost, is_included)`
- Composite primary key `(place_id, amenity_id)`
- Foreign keys to `places.id` and `amenities.id`
- Indexes on both foreign keys

Business Logic Enforcement

- Amenity must be active to be assigned
- Pricing rules:
- `is_included = true → additional_cost = 0`
- Optional amenities add surcharges
- Seasonal or conditional availability checks
- Categorization supports filtering and UI grouping

2.5 User → Booking (1 to Many)

Meaning

A User may make multiple Bookings over time, each for a single Place.

Database Enforcement

- `bookings.user_id → users.id` (foreign key)
- Index on `user_id`
- Constraints:
- `guest_count > 0`
- `end_date > start_date`

Business Logic Enforcement

- Booking eligibility checks:
- Email verification
- Valid payment method
- Acceptable cancellation history
- Booking limits (e.g., max 3 active bookings)
- Age verification for restricted properties
- Optional requirement to review previous stays

2.6 Place → Booking (1 to Many)

Meaning

A Place may receive many Bookings over time, but none may overlap.

Database Enforcement

- `bookings.place_id → places.id` (foreign key)
- Index on `place_id`
- Exclusion constraint preventing overlapping date ranges
- Check constraint: `guest_count ≤ places.max_guests`

Business Logic Enforcement

- Availability window enforcement (e.g., max 12 months in advance)
- Minimum stay length enforcement
- Blackout date support
- Dynamic pricing based on:
- Seasonality
- Demand
- Length of stay
- Preparation buffer between bookings

---

3. Sequence Diagrams for API Calls

Overview

This section illustrates how API requests propagate through the **Presentation**, **Business Logic**, and **Persistence** layers. Each sequence diagram demonstrates:

- Control flow across components
- Validation and authorization checkpoints
- Repository interactions
- Response propagation

These diagrams act as **execution blueprints** for implementing endpoints.

3.1 User Registration

Use Case

Register a new user account in the system.

Detailed Flow

1. Client sends `POST /register` with credentials
2. API Controller:
3. Validates request schema
4. Hashes password
5. API calls `UserService.register()`
6. Service delegates to `HbnbFacade.create_user()`
7. Facade:

8. Checks email uniqueness via `UserRepository`
9. Creates User entity
10. Sets default role and inactive status
11. Repository inserts User into database
12. Facade triggers verification email workflow
13. API returns success response

Design Rationale

- Prevents duplicate accounts
- Separates transport validation from business validation
- Centralizes onboarding workflow

3.2 Place Creation

Use Case

Host creates a new place listing.

Detailed Flow

1. Client sends `POST /places`
2. API:
3. Authenticates token
4. Validates payload
5. API calls `PlaceService.create_place()`
6. Service delegates to `HbnbFacade.create_place()`
7. Facade:
8. Verifies host existence and eligibility
9. Creates Place entity in `draft` state
10. Validates attributes
11. PlaceRepository persists Place
12. API returns created place data

Design Rationale

- Prevents unauthorized listing creation
- Ensures incomplete listings are not publicly visible

3.3 Review Submission

Use Case

Guest submits a review for a completed stay.

Detailed Flow

1. Client sends `POST /reviews`

2. API:
3. Authenticates user
4. Validates payload
5. API calls `ReviewService.create_review()`
6. Service delegates to `HbnbFacade.create_review()`
7. Facade:
8. Confirms place exists
9. Confirms booking exists and belongs to user
10. Confirms review window validity
11. Creates Review entity
12. ReviewRepository persists Review
13. Triggers rating recalculation for Place
14. API returns success response

Design Rationale

- Enforces review authenticity
- Protects rating integrity

3.4 Fetching a List of Places

Use Case

User searches for places using filters (location, price, amenities, dates, etc.).

Detailed Flow

1. Client sends `GET /places?filters`
2. API:
3. Parses query parameters
4. Validates filter schema
5. API calls `PlaceService.search()`
6. Service delegates to `HbnbFacade.search_places()`
7. Facade:
8. Queries PlaceRepository with filters
9. For each Place, loads related Amenities
10. Applies availability and pricing transformations
11. API serializes and returns results

Design Rationale

- Keeps querying logic in persistence layer
- Keeps transformation logic in business layer
- Prevents controllers from embedding domain rules

---

4. Summary

This document consolidates:

- A **layered architectural blueprint**
- A **rich domain model** with explicit business rules
- **Precise relationship constraints** with database and logic enforcement
- **Execution-level API flows**

Together, these components form a complete, production-ready technical specification for implementing the HBnB Evolution backend.

📌 **End of Document**